Measurement of Embedding Choices on Cryptographic API Completion Tasks

YA XIAO, Virginia Tech, USA
WENJIA SONG, Virginia Tech, USA
SALMAN AHMED, Virginia Tech, USA
XINYANG GE, Databricks, USA
BIMAL VISWANATH, Virginia Tech, USA
NA MENG, Virginia Tech, USA
DANFENG (DAPHNE) YAO, Virginia Tech, USA

In this paper, we conduct a measurement study to comprehensively compare the accuracy impacts of multiple embedding options in cryptographic API completion tasks. Embedding is the process of automatically learning vector representations of program elements. Our measurement focuses on design choices of three important aspects, *program analysis preprocessing*, *token-level embedding*, and *sequence-level embedding*. Our findings show that program analysis is necessary even under advanced embedding. The results show 36.20% accuracy improvement on average when program analysis preprocessing is applied to transfer bytecode sequences into API dependence paths. With program analysis and the token-level embedding training, the embedding *dep2vec* improves the task accuracy from 55.80% to 92.04%. Moreover, only a slight accuracy advantage (0.55% on average) is observed by training the expensive sequence-level embedding compared with the token-level embedding. Our experiments also suggest the differences made by the data. In the cross-app learning setup and a data scarcity scenario, sequence-level embedding is more necessary and results in a more obvious accuracy improvement (5.10%).

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering**; • **Security and privacy** → **Software and application security**;

Additional Key Words and Phrases: neural code completion, embedding, deep learning, neural networks, program analysis, API dependency, cryptography, secure coding, Java

1 INTRODUCTION

Code embedding refers to the process of transforming the program elements into continuous vectors [5, 24, 59]. This transformation is important for deep learning, as the subsequent model training and inference are performed on the embedding vectors [10, 17, 40, 52, 54]. Despite much progress in this area [5, 18, 20, 24, 29, 36, 48, 59], it is still unclear the effectiveness and advantages of different embedding designs. A side-by-side comparison would help one better design neural network based methodologies and harness their power for embedding-based applications.

Our work uncovers the impacts of multiple embedding design choices on the API completion task, a foundational question in AI-based software engineering, through comprehensive comparative experiments. API completion

Authors' addresses: Ya Xiao, Virginia Tech, Blacksburg, USA, yax99@vt.edu; Wenjia Song, Virginia Tech, Blacksburg, USA, wenjia7@vt.edu>; Salman Ahmed, Virginia Tech, Blacksburg, USA, ahmedms@vt.edu>; Xinyang Ge, Databricks, Redmond, USA, aegiryy@gmail.com>; Bimal Viswanath, Virginia Tech, Blacksburg, USA, vbimal@vt.edu; Na Meng, Virginia Tech, Blacksburg, USA, nm8247@vt.edu; Danfeng (Daphne) Yao, Virginia Tech, Blacksburg, USA, danfeng@vt.edu.

1049-331X/2023/10-ART
https://doi.org/10.1145/3625291

aims to predict the next API method given the previous code sequence. It is a basic building block for many software engineering tasks, including code repair and code generation. In our experiments, we choose a specific application scenario, cryptographic API completion. Cryptographic APIs are widely known to be error-prone [1, 30, 42, 55, 57]. Misuses, such as predictable random numbers and insecure hash algorithms, severely threaten software security. Thus, this task is more challenging and not well handled by existing solutions because, beyond correctness, security is also required. By experimenting on these challenging APIs, we observe and report the accuracy impacts of different embedding choices.

There are usually three key steps for training code embedding vectors. First, programs are preprocessed into certain representations (e.g., bytecode, control flow graphs) that contains meaningful features. This is usually achieved by program analysis techniques. Based on the preprocessed representations, a basic embedding training vectorizes every single token by gathering its context information across the entire corpus, which is referred to as *token-level embedding*. Beyond embedding a single token, an extra step could be conducted to produce embedding vectors for a given sequence, which is called *sequence-level embedding* in our paper. It requires an extra sequence model pretraining compared with the basic token-level embedding. Therefore, we identify design choices of the three main aspects – **i)** *program analysis preprocessing*, **ii)** *token-level embedding*, and **iii)** *sequence-level embedding* to compare, as shown in Table 1. Such comparison is missing in the literature and needs to be systematically performed.

Our **first comparison group** focuses on the impacts of program analysis preprocessing. Program analysis is often used to process programs before embedding [4, 9, 23, 58]. This preprocessing is important as it decides what information is used for embedding training. For example, Henkel *et al.* [24] extract symbolic traces for embedding while the state-of-the-art code embeddings (e.g., GraphCodeBERT [22], inst2vec [7]) leverage data flows from graph representations to embed program elements. In our work, we compare three program representations, bytecode, program slices, and API dependence paths, obtained with different program analysis strategies for embedding. We explain why the three representations are selected in Section 3.1.

Our **second comparison group** examines the impacts of token-level embedding. We make comparisons between token-level embedding and the one-hot encoding baseline. One-hot encoding is a basic vectorization approach that indexes N tokens and represents the i-th token by an N-dimensional vector that includes a single 1 at the i-th dimension and 0s for other dimensions. Compared with it, token-level embedding, such as word2vec [31–33], is expected to result in low-dimensional semantic-aware vectors that could benefit the downstream task training. By the experimental comparison, we observe how much accuracy improvement the token-level embedding can gain.

Our **third comparison group** learns the impacts of sequence-level embedding (also called contextualized embedding). We make comparisons between sequence-level embeddings and token-level embeddings. Compared with token-level embedding, sequence-level embedding is more advanced because the polysemy issue is handled, by assigning different vectors for different occurrences of a token. However, it also requires an extra expensive sequence language model and pretraining process to achieve that. For example, the state-of-the-art natural language sequence-level embedding BERT [17] is obtained by pretraining the Transformer [51] neural network. Our experimental comparisons aim to answer at what level the advantage of sequence-level embedding is over token-level embedding. Fig. 2 concludes the workflow how we generate the one-hot vectors, token-level embeddings, and sequence-level embeddings.

To evaluate embeddings with different design choices, we perform API completion tasks on our Java cryptographic API benchmark. Our benchmark is composed of Java cryptographic code collected from 79,887 Android apps. To ensure verifiability and reproducibility, our Java cryptographic API benchmark is publicly available on GitHub [1].

[1] https://github.com/Anya92929/DL-crypto-api-auto-recommendation

Next, we explain our research questions along with the comparative experiments designed to answer them.

RQ1: What are the accuracy impacts of token-level embedding obtained from bytecode, slices, and API dependence paths in cryptographic API completion, respectively? To answer this question, we pretrain three token-level embeddings, *byte2vec*, *slice2vec*, and *dep2vec* on bytecode, slices, and API dependence paths, respectively. Bytecode, program slices, and API dependence paths are the outcome of different program analysis preprocessing. The obtained embeddings are compared with the basic setting, one-hot encoding, with corresponding program analysis preprocessing.

RQ2: What are the accuracy impacts of sequence-level embedding obtained from bytecode, slices, and API dependence paths in cryptographic API completion, respectively? To answer this question, we pretrain three sequence-level embeddings, *byteBERT*, *sliceBERT*, and *depBERT* on bytecode, slices, and API dependence, respectively. They are finetuned for the cryptographic API completion and compared with an identical Transformer neural network without the pretraining knowledge.

RQ3: Are our embeddings effective for cryptographic API completion on new apps? To answer this, we perform the experiments not only under the basic within-app setting, but also under the cross-app setting. In the within-app setting, sequences are extracted from Android apps and randomly split for training and testing. In the cross-app setting, new Android apps are used to test the model.

RQ4: How well does the state-of-the-art general purpose code embedding work for cryptographic API completion? Besides the program analysis and embedding choices we covered in Table 1, we further evaluate two state-of-the-art code embedding GraphCodeBert [22] and CodeBert [20] for cryptographic API completion. They are two general purpose source code embedding models pretrained by Microsoft on six programming languages paired with natural language. We finetune the two pretrained models for our API completion task and form an end-to-end comparison.

Our major findings include:

- Our findings show that program analysis preprocessing plays a significant role in cryptographic API embedding and completion. For both token-level embedding and sequence-level embedding, the API dependence paths produce higher prediction accuracy, compared with slices and bytecode. With program analysis, the token-level embedding *dep2vec* achieves an accuracy 36% higher than *byte2vec*. The sequence-level embedding *depBERT* achieves an accuracy 45.86% higher than *byteBERT* without program analysis preprocessing.
- Our findings show that applying embeddings with program analysis significantly improves task accuracy compared with the one-hot baseline (no embedding). On dependence paths, the token-level embedding *dep2vec* and sequence-level embedding *depBERT* both outperform the one-hot encoding baseline by the accuracy boost of 6% and 7%, respectively. Although sequence-level embedding is slightly (0.55%) better than token-level embedding in our experiments. Considering the expensive cost of sequence-level embedding, token-level embedding is more desirable.
- Our findings show that the improvements derived from program analysis and embedding are valid for cryptographic API completion on new apps. In the cross-app learning scenario, the program analysis guided embedding *depBERT* and *dep2vec* still achieve good accuracy at 95.75% and. 93.58%, respectively. Another observation is the advantage of *depBERT* over *dep2vec* is slightly more obvious by the 2.17% accuracy boost compared with 0.55% in the basic setting. The sequence-level embedding *depBERT* is most recommended to be used in the data scarce situation, as the largest improvement (5.10%) of *depBERT* compared with *dep2vec* is observed on the smallest task dataset with 26,357 dependence paths.
- The state-of-the-art general purpose source code embedding solutions GraphCodeBert and CodeBert are insufficient in our cryptographic API completion tasks with a low accuracy of 59.94%. Experiments still show the advantage of applying program analysis preprocessing in their embedding solutions. GraphCodeBert

substantially outperforms its non-program-analysis counterpart CodeBert by an accuracy boost of 20.07% on average. The experiments also suggest the method-level context is more recommended than the class-level context for cryptographic API completion.

Significance of research contributions. Our work provides the first quantitative and systematic comparison of the prediction accuracy of multiple API embedding approaches for neural network based code completion. Our rigorous experiments provide new empirical results that have not been previously reported, including how various domain-specific program analyses improve data-driven predictions. These quantitative findings help guide and design more powerful and accurate code completion solutions, leading to high quality and low vulnerability software projects in practice. As cryptographic API completion is more difficult and requires a deeper understanding of the code context, we expect our observations to be valid and useful for general code completion tasks as well. We keep the general evaluation as our further work. We also publish our new cryptographic API benchmark along with our deep learning models to help future research.

2 BACKGROUND

We provide the background of embedding and the cryptographic API completion task. We categorize embedding vectors into token-level embeddings and sequence-level embeddings.

2.1 Token-level Embedding

Token-level embeddings, such as word2vec [31–33], FastText [8, 25], Glove [39], assign one numeric vector to represent a token. In our work, we follow the skip-gram [31, 32] algorithm to train token-level embeddings for API methods and constants. Specifically, a three-layer linear neural network is used to automatically learn the embeddings of all tokens in an embedding sequence corpus. The token (API method or constant) to be embedded is the input, and the tokens before and after it within a sliding window are used as the labels to train the neural network. During the embedding process, the entire embedding corpus is scanned and all the tokens and their neighbors are used for training. After that, the latent vector at the hidden layer is kept as the embedding of the input token. In this way, a token's embedding vector is determined by the statistics of its neighboring tokens in a large corpus.

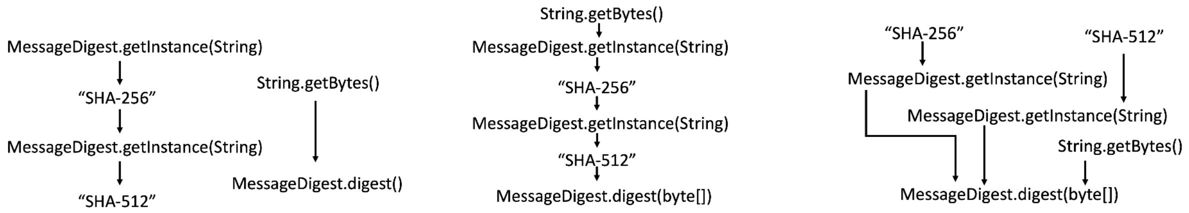
2.2 Sequence-level Embedding

Sequence-level embedding assigns a vector for every occurrence of a token. In other words, a token is represented with different vectors when it appears in different sequences. To generate this contextualized vector, not only the token itself but also other tokens in a given sequence are used. There is a neural network based language model to take a sequence as input and output the embedding vectors of every token in this sequence. For example, the GPT family [41], BERT [17], RoBERTa [28], are sequence-level embeddings generated from Transformer neural networks. The sequence-level embedding ELMo [40] is generated from a BiLSTM neural network. This neural network is pretrained with carefully crafted tasks for producing the sequence-level embedding. Therefore, it is also referred to as a pretrained language model. The sequence-level embedding of a token is dynamically generated by the pretrained language model.

To apply the sequence-level embeddings for downstream tasks, a common way is to use the pretrained model that produces sequence-level embedding as the initial states. An extra application layer is added after the embedding layer and the entire model is fine-tuned with extra data for a specific downstream task.

2.3 Cryptographic API Completion

We evaluate different embeddings in cryptographic API completion. API completion refers to a task that suggests one or more next API methods given a preceding sequence of API elements (i.e., *API methods and constants*). We


MessageDigest.getInstance(String)
"SHA-256"
MessageDigest.getInstance(String)
"SHA-512"
String.getBytes()
MessageDigest.digest()
String.getBytes()
MessageDigest.getInstance(String)
"SHA-256"
MessageDigest.getInstance(String)
"SHA-512"
MessageDigest.digest(byte[])
"SHA-256"
"SHA-512"
MessageDigest.getInstance(String)
MessageDigest.getInstance(String)
String.getBytes()
MessageDigest.digest(byte[])

(a) Two bytecode sequences (b) Slice sequence (c) API dependence graph

Fig. 1. API and constant sequences from bytecode, program slices and API dependency graphs

define two types of cryptographic API completion tasks, i.e., *next API completion* and *next API sequence completion*. The former aims to predict one API method in the next line while the latter produces a sequence of API methods to invoke sequentially.

3 OUR MEASUREMENT SETTING

We perform comparative experiments to answer our research questions. As shown in Table 1, we compare different design choices of *program analysis preprocessing*, *token-level embedding*, and *sequence-level embedding*.

Table 1. The overview of our comparative settings. Each cell shows the embedding and machine learning model we use for the cryptographic API completion. We have comparisons between three program analysis preprocessed sequences, token-level embeddings vs. one-hot, and sequence-level embeddings vs. token-level embeddings.

	Program analysis preprocessing		
	Bytecode	Program slices	API dependence paths
Token-level embedding	byte2vec vs. one-hot (w/ LSTM)	slice2vec vs. one-hot (w/ LSTM)	dep2vec vs. one-hot (w/ LSTM)
Sequence-level embedding	byteBERT vs. byte2vec (w/ Transformer)	sliceBERT vs. slice2vec (w/ Transformer)	depBERT vs. dep2vec (w/ Transformer)

3.1 Program Analysis Preprocessing Strategies

We examine the impacts of using program analysis to guide the embedding. There could be unlimited program analysis strategies to extract different program sequences. Specifically, we compare three types of program sequences: *i) bytecode*, *ii) program slices*, and *iii) API dependence paths*. The bytecode is from Android apps without program analysis. The program slices are obtained by conducting interprocedural backward slicing on bytecode. Moreover, the API dependence paths are extracted from API dependence graphs we construct on program slices with the dataflow dependence between the API calls. We select these three because they embody the increasing levels of program analysis guidance.

Bytecode sequences. We extract the API sequences directly from the Android bytecode. For each method implementation, we extract the API methods and constants used in it into one sequence. There is no ordering between sequences collected from different method implementations. Based on our observation, the order of the API methods and constants in these sequences is close to their order in the source code. We cover the bytecode option because it reflects the effect of embedding without program analysis guidance.

Program slices. We apply a program analysis strategy, *interprocedural backward slicing*, to obtain program slices. The slicing starts from the variables used with a cryptographic API invocation. By backwardly tracing the data flows reaching these variables, all the code statements influencing the API invocation are kept while irrelevant code statements are excluded. When reaching the entry point of the current method, we jump to its callers to continue the backward tracing until the tracked data facts are empty or there is no caller found. In this way, the influencing code context beyond a local method is also collected. When meeting a self-defined method (i.e., a method that is written by the developer and is not provided by Java libraries) call, we replace it with its implementation code if available. An example of program slices is shown in Fig. 1b. A major difference between program slices and bytecode is that the irrelevant predecessors are removed by program analysis.

API dependence paths. With program analysis, the code semantic information, such as program dependencies, can be extracted. We perform the *API dependence graph construction* and extract the API dependence paths for embedding. The API dependence graphs are built through dataflow analysis. We add the data dependence edges between API calls on slices. An example of our API dependence graph is shown in Fig. 1c. It uses an API or a constant as a node. Two nodes having the data dependence (`def-use`) relationship are connected directly. The API dependence paths are covered in our measurement as a representative of the state-of-the-art code semantic based approaches. [7, 22].

Experimental setup of program analysis preprocessing. We implement an interprocedural, context- and field-sensitive dataflow analysis to achieve our backward slicing and API dependence graph construction. The analysis is implemented with the Java program analysis framework Soot [50]. Soot takes the Android bytecode as input and transforms it into an intermediate representation (IR) Jimple. The program analysis (i.e., slicing or API dependence graph construction) is performed on Jimple IR. We use Soot 2.5.0, Java 8, and Android SDK 26.1.1.

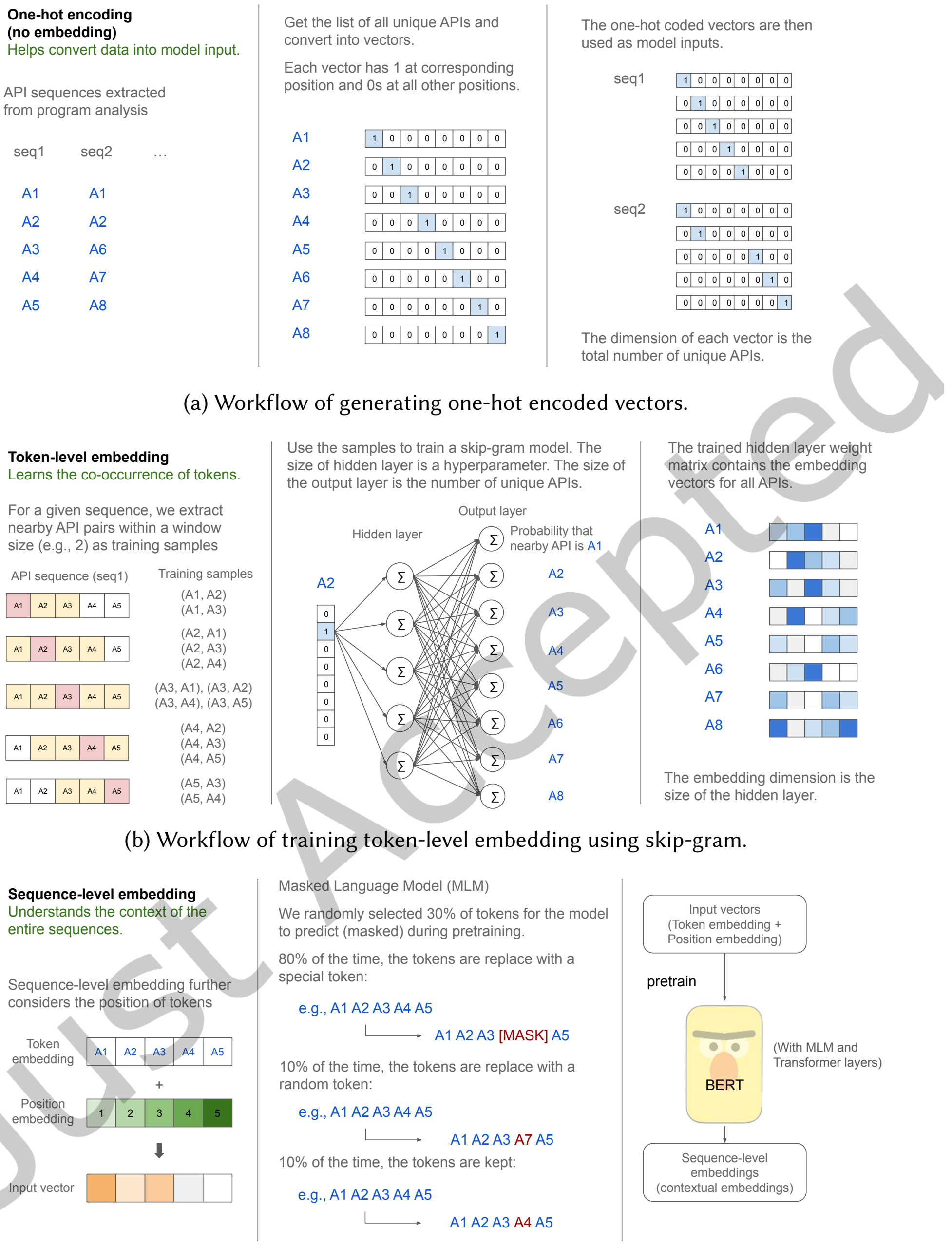
3.2 Token-level Embedding Settings

We perform the token-level embedding training to produce vectors for tokens in an embedding vocabulary, as illustrated in Fig. 2b.

Cryptographic code identification. All the embeddings are produced from the cryptographic code corpus we extract from decompliled Android Apks. We refer to the code implemented with cryptographic API calls as cryptographic code. To identify cryptographic code from an Android App, we first search all the cryptographic API callsites within the codebase. All the method signatures within the Java package `java.security` and `javax.crypto` (see Table 2) are included in our search list. Then, we start from these cryptographic API callsites to find other standard API calls happening **before** a cryptographic API callsite as its context. However, there might be different accurate levels and scopes of the context according to preprocessing. In bytecode sequences, we can only extract all the previous API calls within the same method of a cryptographic callsites as its context. When program analysis technique is applied, we are able to generate more meaningful API call context based on its program dependency. In our experiments, a cryptographic API callsite and its program-wide dependency code is extracted as an inter-procedural (cross-method) program slice. The entire slice are regarded as cryptographic code and all the previous API calls within this slice will be gathered as the context of a cryptograhpic API call.

Embedding vocabulary. The embedding vocabulary is collected during the cryptographic code identification. The vocabulary initially includes the standard JAVA cryptographic APIs. Then, we scan the App and perform interprocedural backward slicing from the detected cryptographic API callsites as entry points. In this way, the vocabulary expands with all the encountered API calls and constants during backward program slicing. When an API call is encountered, we first check whether it is a self-defined method[2] If it is, the analysis jumps into the implementation of this method according to the interprocedural slicing algorithm. Otherwise, the API method will be collected as an element in our vocabulary. For the collected API methods, we further filter those that appear less than five times. For constants, we manually identified 104 reserved string constants used as the arguments of cryptographic APIs. Other constants that appear more than 100 times in the slices are also kept in the embedding vocabulary. Finally, we have a vocabulary of 4,543 tokens (3,739 APIs and 804 constants). The

[2] The method defined and implemented by developers within this program.



One-hot encoding (no embedding)
Helps convert data into model input.

API sequences extracted from program analysis

Get the list of all unique APIs and convert into vectors.

Each vector has 1 at corresponding position and 0s at all other positions.

The one-hot coded vectors are then used as model inputs.

The dimension of each vector is the total number of unique APIs.

(a) Workflow of generating one-hot encoded vectors.

Token-level embedding
Learns the co-occurrence of tokens.

For a given sequence, we extract nearby API pairs within a window size (e.g., 2) as training samples

Use the samples to train a skip-gram model. The size of hidden layer is a hyperparameter. The size of the output layer is the number of unique APIs.

The trained hidden layer weight matrix contains the embedding vectors for all APIs.

The embedding dimension is the size of the hidden layer.

(b) Workflow of training token-level embedding using skip-gram.

Sequence-level embedding
Understands the context of the entire sequences.

Sequence-level embedding further considers the position of tokens

Masked Language Model (MLM)

We randomly selected 30% of tokens for the model to predict (masked) during pretraining.

80% of the time, the tokens are replace with a special token:

e.g., A1 A2 A3 A4 A5 → A1 A2 A3 [MASK] A5

10% of the time, the tokens are replace with a random token:

e.g., A1 A2 A3 A4 A5 → A1 A2 A3 A7 A5

10% of the time, the tokens are kept:

e.g., A1 A2 A3 A4 A5 → A1 A2 A3 A4 A5

Token embedding
A1 A2 A3 A4 A5
+
Position embedding
1 2 3 4 5
Input vector
Input vectors (Token embedding + Position embedding)
pretrain
BERT
(With MLM and Transformer layers)
Sequence-level embeddings (contextual embeddings)

(c) Workflow of training sequence-level embedding using BERT.

Fig. 2. Workflow of generating one-hot encoded vectors, token-level embedding, and sequence-level embedding. Here we aim to show the high-level idea of the process and what information each type of embedding carries. We refer interested readers to the original paper of Word2Vec [32] and BERT [17] for technical details.

API methods include the standard APIs from Java and Android platforms, as well as some third-party APIs that cannot be inlined because of recursion or phantom methods (whose bodies are inaccessible during the analysis). Table 2 shows the library distribution of these API methods.

Table 2. The library sources of our embedding APIs

Source		**# of embedded APIs**
Java platform	java.security	510
	javax.crypto	166
	java.io	138
	java.lang	259
	others	374
Android platform		486
Third parties		1,827

We train the skip-gram embedding model [32] to obtain the word2vec-like embedding. With different program analysis preprocessing, three types of token-level embeddings, *byte2vec*, *slice2vec*, and *dep2vec* are produced.

- *byte2vec* is the baseline embedding version that applies *word2vec* [31, 32] directly on the bytecode corpus.
- *slice2vec* is the embedding with the inter-procedural backward slicing as the pre-processing method.
- *dep2vec* applies API dependence graph construction to guide the embedding training.

Experimental setup for token-level embeddings. We follow the convention of the natural language embedding word2vec to set hyperparameters. The embedding vector length is 300. The sliding window size for neighbors is 5. We also applied subsampling and negative sampling to randomly select 100 false labels to update in each batch. Based on our preliminary experiments, we train embeddings with a mini-batch size of 1024. The embedding terminates after 10 epochs. Because we did not observe significant improvement by longer epochs and smaller batch size. Our embedding model is implemented using Tensorflow 1.15. Training runs on the Microsoft AzureML GPU clusters, which support distributed training with multiple workers. We use a cluster with 8 worker nodes. The VM size for each node is the (default) standard NC6.

3.3 Sequence-level Embedding Settings

We obtain sequence-level embeddings by applying the method of training the well-known natural language embedding BERT [17] on program sequences, as shown in Fig. 2c.

byteBERT vs. sliceBERT vs. depBERT. On bytecode, program slices, and API dependence paths, we obtained three different versions of BERT embeddings for API elements, *byteBERT*, *sliceBERT*, and *depBERT*. These BERT-like API embeddings are produced by pretrained Transformer neural networks. We apply the masked language modeling (MLM) task to pretrain them. MLM is a task that reconstructs language sequences with masked tokens. It predicts the missing tokens for a given sequence with random masks. The masked tokens in the input sequence are either replaced by a special token [MASK] or an arbitrary random token in the vocabulary or kept in original sequences. We set the probabilities of the three situations as 80%, 10%, and 10%, and follow the convention in NLP. The masked tokens are randomly selected with a probability of 30% and one sequence is limited to having two masked tokens at most. These are similar to the setting of the MLM for training BERT [17]. We discard the next sentence prediction (NSP) of BERT as there is no corresponding concept of the “next sentence” between two code sequences. Three types of sequence-level embeddings are trained with identical hyperparameters. Same with the LSTM training with token-level embedding, the neural network is trained for 10 epochs with a batch size of 1024. When training the Transformer model, the input tokens are represented as our token-level embedding. To apply

Table 3. Overview of our datasets

Dataset	App Set ID	Apps	Experiments
Basic	1	16,048	Embedding API Completion
Advanced	2	64,478	Embedding
	3	11,997	API Completion
	4	1,819	API Completion
	5	1,055	API Completion
	6	538	API Completion

these sequence-level embedding in cryptographic API completion, the pretrained neural networks are finetuned by the given task-specific data.

Training parameters. Due to the resource constraint, we cannot thoroughly try every possible parameter combination (grid search). Instead, we select the optimal parameters with some preliminary experiments. We considered different numbers of epochs (up to 20), batch sizes (512, 1024), and learning rates (0.1, 0.01, 0.001). Choosing different learning rates improves the accuracy by no more than 0.02. After training for 10 epochs, the accuracy increment is less than 0.001 for each extra epoch. Therefore, we chose the final set of parameters (10 epochs, 0.001 learning rate, and 1024 batch size) to achieve a balance between computational resources and model performance.

For the pretraining of our byteBERT, sliceBERT, and depBERT models, we choose the ratio of masking strategies (80%, 10%, and 10%) following the original BERT paper. We choose the mask ratio of 30% because some of the sequences are short and we want each sample having one or two masked tokens. As those parameters are optimized by the authors of BERT, they are also the most commonly used setting in the NLP field.

Differential evolution (DE) is not a common practice in choosing hyperparameters for deep learning models, while it is more frequently used for tuning kernel parameters for SVM. Applying DE on top of LSTM models and evaluating its impact on model performance can form an interesting research topic by itself. We leave this extension as our future research direction.

3.4 Dataset Overview

We conduct experiments on Android apps collected from the Google Play store. We choose the Android platform because of its widespread use and popularity among users. We collect apps from various categories to ensure the dataset reflects a diverse usage of Java cryptographic API in practice. According to the way we split data for training and testing, we have a basic dataset and an advanced cross-app data setup. Table 3 gives an overview.

3.4.1 Basic Data Split Setting. The basic dataset is composed of 16,048 Android apps from three categories, 5,176 apps from the business category, 4,581 Apps from the communication category, and 6,291 apps from the finance category. From these apps, we extracted 707,775 API sequences from bytecode, 926,781 API sequences from program slices, and 566,279 API sequences from API dependence graphs. The number of tokens in the three types of sequences is shown in Table 4. The tokens refer to the APIs or constants in our embedding vocabulary.

Table 4. Embedding corpora statistics of the basic dataset

Corpora	Bytecode	Slices	Dependence paths
# of tokens	28,887,852	12,341,912	38,817,046

For embedding, we use all of the API sequences to produce the token-level embeddings and sequence-level embeddings. For API completion tasks, we randomly split all the sequences for training and testing following the ratio of 4:1.

3.4.2 Advanced Data Split Setting. We create an advanced dataset to enable cross-app learning and validate our findings on new apps. Under this setup, the collected apps are split for embedding and API completion tasks, respectively. This guarantees that the apps used for API completion tasks are not seen in the embedding training phase. Our embedding experiments are conducted on 64,478 apps (app set 2), which are much more than the app sets 3, 4, 5, and 6 that we used for API completion tasks. This consideration is because embeddings are often pretrained with huge data volumes and released for fine-tuning with smaller task-specific datasets in the real world. Then, the apps for API completion tasks are split into training and testing sets. This guarantees that the apps used for testing are never seen in the training. Compared with the basic dataset, the cross-app setting is more practical and challenging. It evaluates whether the model trained on a set of apps can be applied to new apps.

In addition, to observe the impacts of the task data volume, we perform API completion training and testing on four app sets (app sets 3, 4, 5, 6) varying in data sizes. The largest one, App set 3, is a diverse App set including 11,997 apps from 12 App categories. Besides, there are three smaller App sets (App sets 4, 5, 6) consisting of 1,819 apps from the personalization category, 1,055 apps from the social category, and 538 apps from the weather category.

Data duplication. For both the basic and advanced dataset, we deduplicate the data in the class file level to guarantee that the reused class files (e.g., libraries) only appear once when extracting the bytecode sequences. However, we did not deduplicate the program slices and the API dependence paths extracted by program analysis. The presence of duplicate slices or paths in the training set suggests common coding patterns. The frequency of API occurrence helps the embedding model learn their relationship. Different source code could follow a similar cryptographic function usage pattern in some cases, as many security principles do not change for various scenarios. Let the model directly learn the processed highly frequent sequences can significantly reduce the expensive data size and training resource requirement. Moreover, since the apps we collected are all real-world apps, this duplication should also hold for apps in the wild and will not affect the performance after deployment. In the cross-app experiments, our goal is to show the model does not make predictions because of the duplicated code sequences from the same app. If a usage pattern is general across multiple apps, it is reasonable to keep their duplicated occurrence in our dataset [2].

4 EVALUATION RESULTS

In this section, we report the accuracy of the cryptographic API completion to compare the impacts of different embedding choices and answer our research questions (RQs). In the evaluation, we calculate top-1 accuracy that only considers the correctness of the top-1 prediction of the model. It is calculated as the number of correct top-1 predictions over the total number of predictions. The top-1 prediction is considered correct if it matches the ground truth from the sequence itself. For API dependence paths from a graph, there might be multiple correct answers due to the branches of the graph.

4.1 Performance Improvement from Token-level Embedding (RQ1)

The impact of applying token-level embedding (RQ1) is measured by comparing it with one-hot encoding on bytecode, slices, and dependence paths, respectively. The accuracies of the API completion tasks are shown in Tables 5 and 6 and a comparison is shown in Figure 3. We visualize the accuracy differences brought by various design choices, namely applying token-level embedding, program analysis preprocessing (i.e., program slicing, and API dependence graph build), or increasing the model sizes in Figure 8 in the appendix.

Experimental setup for cryptographic API completion tasks. We train LSTM based models for the task. For *next API completion* task, we train the LSTM based sequence model to accept a sequence of API methods or constants $(t_1, t_2, \ldots, t_{n-1})$ and output the next API t_n. For *next API sequence completion* task, we train the LSTM based seq2seq (encoder-decoder) model to accept the first half API sequence $(t_1, t_2, \ldots, t_n)$ and predict the last half of the sequence $(t_{n+1}, t_{n+2}, \ldots, t_{2n})$.

We filter our code dataset using CryptoGuard [42], which is a static cryptography API misuse detection tool. We exclude insecure cryptographic API usage to prevent the embeddings and models from learning these vulnerable patterns [44]. It also helps eliminate the situations that when the models predict secure APIs and the ground truth itself (from the original data) is insecure, such predictions are counted as incorrect answers. This step contributes to a more accurate and meaningful evaluation of model performance. We limit the maximum number of LSTM steps to 10. We use a batch size of 1,024 and a learning rate of 0.001. The highest accuracy achieved within 10 epochs is recorded. These hyperparameters are selected because no obvious accuracy improvement is observed by longer epochs, smaller batch size or learning rate. We use the stacked LSTM architecture with vanilla LSTM cells for the LSTM-based models.

Table 5. Top-1 accuracy of the next API token completion on the basic dataset.

LSTM	Bytecode		Slices		Dependence Paths	
Units	1-hot	byte2vec	1-hot	slice2vec	1-hot	dep2vec
64	49.78%	48.31%	66.39%	78.91%	86.00%	86.33%
128	53.01%	53.52%	68.51%	80.57%	84.81%	87.75%
256	54.91%	54.59%	70.35%	82.26%	84.57%	91.07%
512	**55.80%**	**55.96%**	**71.78%**	**83.35%**	**86.34** %	**92.04%**

Table 6. Accuracy of the next API sequence completion on the basic dataset. We use the LSTM-based sequence model with a hidden layer size of 256 for this task.

Bytecode		Slices		Dependence Paths	
1-hot	byte2vec	1-hot	slice2vec	1-hot	dep2vec
43.61%	44.63%	64.10%	85.02%	82.94%	89.23%

4.1.1 *Bytecode vs. program slices vs. API dependence paths.*

Tables 5 and 6 show the accuracy results of the *next API completion* and the *next API sequence completion*, respectively. To uncover the impact of the program analysis preprocessing, both the token-level embedding (i.e., *byte2vec*, *slice2vec*, *dep2vec*) and the one-hot encoding baseline are used to train the LSTM models on bytecode, slices, and dependence paths.

We observe that program analysis preprocessing shows significant benefits. Table 5 shows the accuracy based on dependence paths is 92%, which is 9% and 36% higher than using slice- and bytecode-based token-level embedding, respectively. The API completion accuracy with one-hot encoding is also substantially improved by program analysis. The accuracy with one-hot encoding increases from 56% on bytecode to 72% on slices, and further to 86% on dependence paths. The results of the next API sequence completion (Table 6) are also consistent with the conclusion. It shows that the accuracy achieved with *byte2vec* improved by 40.39% with *slice2vec*, and improved by 44.60% with *dep2vec*.

[3] dep2vec column - byte2vec column in Table 5

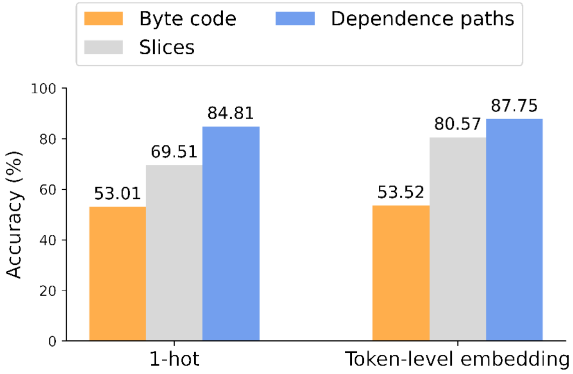


Fig. 3. Comparison of accuracy of with and without token-level embedding and applying different program analysis preprocessing (based on LSTM-128). The group on the left are results without embedding. The group on the right are results with token-level embedding. Applying token-level embedding improves the model performance in all three cases. The three colors represent three different program analysis approaches. With or without embedding, the dependence path outperforms the other two strategies and the slice outperforms bytecode. Compared to applying embedding, program analysis preprocessing boosts performance more.

Finding 1: For Crypto API completion with token-level embeddings, program analysis significantly improves the accuracy by 36.20%[3] on average.

4.1.2 Token-level embedding vs. one-hot vectors. On each program analysis preprocessing representation, we compare the token-level embedding and the one-hot encoding baseline. We observe significant improvements by applying token-level embeddings on slices and dependence paths. However, the improvement in bytecode is limited. Table 5 shows that *slice2vec* improves the accuracy by 11% from its one-hot baseline. *dep2vec* improves the accuracy by 6% from its one-hot baseline. These improvements suggest that *slice2vec* and *dep2vec* capture useful information. This conclusion is also observed in the next API sequence recommendation task. *slice2vec* and *dep2vec* improve the accuracy from their baselines by around 21% and 6%, respectively. In contrast, *byte2vec* does not show any significant improvement from its one-hot baseline.

Finding 2: For cryptographic API completion on program slices and API dependence paths, token-level embedding achieves an average accuracy improvement of 12.02% and 3.97%, respectively, compared to one-hot vectors.

We also observe higher accuracy achieved by longer LSTM units, which is as expected. However, the accuracy benefits gained by increasing the model size from LSTM-64 to LSTM-128, from LSTM-128 to LSTM-256, and from LSTM-256 to LSTM-512, are smaller and smaller.

Overall, the best accuracy is achieved by *dep2vec* in both tasks, an accuracy of 92.04% in the *next API completion* task and an accuracy of 89.23% in the *next API sequence completion* task. Compared with the basic one-hot encoding on bytecode (no program analysis preprocessing), they achieve substantial accuracy improvements (36% and 46%, respectively) in both tasks. Although all the measures, including token-level embedding, program analysis preprocessing, and increasing model sizes, improve the accuracy, the two program analysis preprocessing

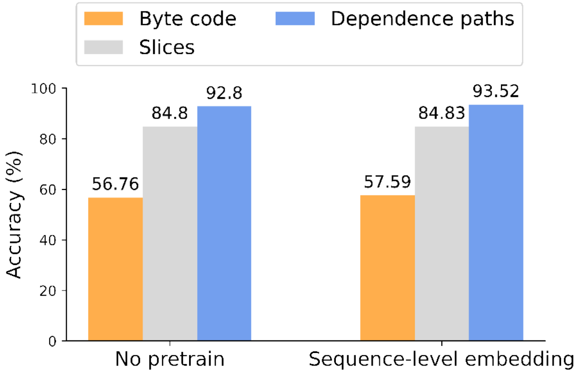


Fig. 4. Comparison of accuracy of with and without sequence-level embedding and applying different program analysis preprocessing (based on Transformer-base). The group on the left are results without sequence-level embedding (no pretrain). The group on the right are results with sequence-level embedding (pretrained). Pretraining sligtly improves the model performance in all three cases. The three colors represent three different program analysis approaches. With or without pretraining, the dependence path outperforms the other two strategies and the slice outperforms bytecode. Compared to applying pretraining, program analysis preprocessing boosts performance more.

strategies, program slicing, and API dependence graph construction, are most effective, resulting in 22.03% and 12.10% accuracy differences on average, respectively.

4.2 Performance Improvement from Sequence-level Embedding (RQ2)

Next, we evaluate the effectiveness of sequence-level embedding (RQ2) from the comparison with token-level embedding on bytecode, slices, and dependence paths, respectively. We fine-tune the sequence-level embedding (i.e., *byteBERT*, *sliceBERT*, or *depBERT*) with the task-specific training before applying them to the API completion task. Then, the models are compared with unpretrained Transformer networks with token-level embeddings. We use two Transformer neural networks with different sizes, namely Transformer-base and Transformer-small. The Transformer-base model has 12 hidden layers with size 768 and 12 attention heads. The Transformer-small model has 4 hidden layers with size 512 and 4 attention heads. Results are shown in Tables 7. We also show a comparison in Figure 4 and visualize the accuracy differences in Figure 9 in the appendix.

Table 7. Accuracy of the next API completion with or without sequence-level embedding (pretrain) on the basic dataset.

Model Size	**Bytecode**		**Slices**		**Dependence Paths**	
	Transformer + byte2vec (w/o. pretrain)	byteBert (w. pretrain)	Transformer + slice2vec (w/o. pretrain)	sliceBert (w. pretrain)	Transformer + dep2vec (w/o. pretrain)	depBert (w. pretrain)
Small	44.38%	45.21%	83.37%	84.15%	90.96%	91.07%
Base	56.76%	57.59%	84.80%	84.83%	92.80%	93.52%

4.2.1 *Bytecode vs. program slices. vs. API dependence paths.*

Table 7 shows that program analysis preprocessing is still necessary even with sequence-level embeddings. The accuracy of using bytecode sequences is low (45.21% and 57.59%) compared with program slices and API dependence paths. With the program analysis, the small and base Transformer neural networks with *depBERT* achieve the accuracy of 91.07% and 93.53%, respectively. When there is only token-level embedding, this conclusion still holds. The small and base Transformer neural networks with *dep2vec* achieve the accuracy of 90.96% and 92.80%, respectively, which are 46.58% and 36.04% higher than the *byte2vec* on bytecode sequences.

Finding 3: For Crypto API completion with sequence-level embedding, program analysis makes a substantial accuracy improvement of 40.90%[4] on average.

According to Figure 4, we observe the impact of program analysis is still the most significant way to improve the accuracy. The average accuracy differences achieved by program slicing and API dependence path construction are 33.55% and 7.80%, respectively, which are much more effective than the sequence-level embedding and a larger Transformer neural network. In Table 7, the small *depBERT* that has program analysis preprocessing achieves an accuracy of 91.07%, which is 34.31% higher than the larger model without program analysis, namely the base *byteBERT*.

By comparing the Transformer with token-level embeddings in Table 7 and the LSTM with token-level embeddings in Table 5, we found that the LSTM-512 achieves slightly higher accuracy than the Transformer-small with a comparable size (hidden size 512).

Finding 4: For Crypto API completion, LSTM-512 shows a 4.22% [5] accuracy advantage on average over Transformer-small (hidden size 512).

4.2.2 Sequence-level embedding vs. token-level embedding. Sequence-level embeddings only show slight advantages over token-level embeddings. As shown in Table 7, the accuracy trained with the sequence-level embeddings is only slightly higher (0.55% on average) than the Transformer neural network with their token-level baselines. One possible reason for this slight improvement observed may be attributed to the strong learning ability of the Transformer model. Through the use of its attention mechanism, the model can effectively learn and comprehend contextual information, even with limited embedded information in the input (token-level embedding) and no pretraining. Another possible reason is the simplicity of programming languages compared to natural languages. Sequence embedding helps capture the different meanings of the same word in various positions or contexts. One example is the different interpretations of the word "like" in the sentence "I like the way you look like." However, such conditions are less likely to occur in a programming language, leading to a smaller improvement when being applied to programming languages than natural languages. Therefore, considering the cost, sequence-level embedding is not recommended in this case.

Besides, the impact of the neural network size is also more obvious than the impact of applying sequence-level embedding. As shown in Table 7, the base Transformer improves the accuracy by 12.38%, 1.43%, and 1.84%, on bytecode, slices, and dependence paths, respectively, compared with the small Transformer.

Finding 5: Although resulting in slight accuracy improvement (0.55% [6] on average), sequence-level embedding is not the first recommended strategy to improve the cryptographic API completion, compared with program analysis and a larger model.

[4] depBERT column - byteBERT column in Table 7

[5] Compare the Transformer columns in Table 7 with the byte2vec, slice2vec, and dep2vec columns in Table 5

[6] Compare between BERT columns and Transformer columns in Table 7

4.3 Cross-app Evaluation (RQ3)

Cross-app learning is a practical scenario in which we expect a pretrained model can be applied to other projects unseen in the training phase. Therefore, we conduct experiments to verify whether our conclusions still hold for new apps that never appear in the training.

Table 8. Accuracy of the next API completion with or without sequence-level embedding (pretrain) on dependence paths of the advanced dataset (cross-app learning).

App Set	# of cases	Transformer + dep2vec (w/o. pretrain)	depBert (w. pretrain)	Improvement
3	813,737	97.23%	98.24%	1.01%
4	97,224	98.66%	99.54%	0.88%
5	88,143	95.09%	96.78%	1.69%
6	26,357	83.34%	88.44%	5.10%
Ave.	**256,363**	**93.58%**	**95.75%**	**2.17%**

Table 9. Accuracy of the next API completion with or without sequence-level embedding (pretrain) on bytecode sequences of the advanced dataset (cross-app learning).

App Set	# of cases	Transformer + byte2vec (w/o. pretrain)	byteBert (w. pretrain)	Improvement
3	7,275,324	79.72%	80.00%	0.28%
4	814,551	86.91%	87.21%	0.30%
5	840,381	77.46%	77.96%	0.50%
6	220,543	65.05%	67.41%	2.36%
Ave.	**2,287,700**	**77.29%**	**78.15%**	**0.86%**

Tables 8 and 9 show the API completion experiments on our advanced dataset (see Section 3.4) which follows the cross-app learning scenario. App sets 3, 4, 5, and 6 include apps that generate task-specific data. For every app category, we randomly select 80% apps of this category to generate training data and 20% apps to generate testing data. In another word, our training and testing data is cross-app but within a category.

Tables 8 and 9 compare sequence-level embeddings (i.e., *depBERT* and *byteBERT*) with the corresponding token-level embeddings (i.e., *dep2vec* and *byte2vec*). *DepBERT* and *byteBERT* are Transformer neural networks pretrained on app set 2 (see Table 3) with Masked Language Model (MLM). We use the small Transformer neural network for all the experiments. Figure 10 in the appendix shows the accuracy differences achieved by program analysis and sequence-level embedding on App sets 3, 4, 5, and 6, respectively.

We observe similar conclusions with the basic dataset about program analysis. The experiments on API dependence paths (Table 8) again show significant advantages compared with bytecode sequences (Table 9). Program analysis preprocessing makes significant accuracy differences (16.95% on average) in all situations.

A minor difference we observe is that sequence-level embedding brings more obvious improvement than on the basic dataset. As shown in Table 8, the average improvement of applying the sequence-level embedding is 2.17%. This indicates that sequence-level embedding is more significant when we train our models in the cross-app scenario. We observe that the sequence-level embedding substantially improves the accuracy for small data sizes. It achieves an accuracy 5.10% higher than the Transformer with *dep2vec*.

From Table 9, we also observe the improvement of applying sequence-level embedding *byteBERT* on bytecode sequences. However, without program analysis, the improvements (0.86% on average) are quite small.

Finding 6: In the cross-app setting, sequence-level embedding achieves more obvious accuracy improvements (2.17% on average) compared with the basic data split setting. We recommend using sequence-level embedding in cross-app learning when the data size is small.

4.4 Comparison with State-of-the-art (RQ4)

Besides the design choices we covered, we further experiment on two state-of-the-art sequence-level embeddings, GraphCodeBert [22] and CodeBert [20]. GraphCodeBert and CodeBert are general-purpose code embedding models pretrained by Microsoft. They adopt the Transformer-based neural architecture and pretrain it on CodeSearchNet dataset [?] which includes 2.3 million functions of six programming languages paired with natural language description. The differences between them are their code preprocessing parts and sequence-level embedding tasks. CodeBert treats code as a sequence of tokens and is pretrained by masked language modeling (MLM). GraphCodeBert uses program analysis to extract dataflow information as input and is pretrained by two extra structure-aware tasks introduced by the authors.

Table 10 shows the next API completion experiments on our app sets 4, 5, and 6. We decompiled .apk files into source code for the neural network inputs. Although there might be lost information due to obfuscation. However, the amount of information loss caused by obfuscation is equal to our three methods (i.e., bytecode sequences, slices, and dependence paths), CodeBERT, and GraphCodeBert. Therefore, we think it still forms a fair comparison. For each cryptographic API call, we extract two types of source code context for it, the method-level context and the class-level context. The former extracts the previous code within the wrapper method where the target call locates while the latter collects the previous code lines found in the same class of the target call. We finetune the two models with our data for 10 epochs with batch size 16. We use this setting because no substantial improvement is observed by longer epochs or smaller batch sizes.

Finding 7: The state-of-the-art general purpose pretrained models only achieve a low accuracy (59.64% by GraphCodeBert on average) for cryptographic API completion. The program analysis preprocessing and the method-level context are recommended.

Table 10. Accuracy of the next API completion by finetuning the general purpose pretrained model GraphCodeBert and CodeBert.

App	GraphCodeBert		CodeBert	
Set	Method-level Context	Class-level Context	Method-level Context	Class-level Context
4	60.45%	39.87%	41.72%	30.82%
5	64.53%	37.83%	41.29%	31.80%
6	54.84%	35.25%	36.60%	31.32%
Ave.	**59.94%**	**37.65%**	**39.87%**	**31.31%**

We have three observations from Table 10. First, the best accuracy is achieved by GraphCodeBert with the method-level context. However, the accuracy is still at a low level, an average of 59.94%. Second, GraphCodeBert substantially outperforms CodeBert in identical data and context settings. When using method-level context, GraphCodeBert has an accuracy of 20.07% higher accuracy than CodeBert on average. When using class-level context, GraphCodeBert achieves 6.34% higher accuracy on average. This confirms our findings 1 and 3 that program analysis contributes a substantial improvement to the embeddings. Another observation is that method-level context is much better than class-level context. With GraphCodeBert, the method-level context outperforms the class-level context by 22.29% accuracy improvement on average. With CodeBert, the method-level context results in an 8.80% higher accuracy on average. The reason might be that the class-level context includes much more irrelevant information and makes the prediction worse.

Furthermore, with the rapid development of large language models, their application in code completion and code repair has been discussed widely. Recent work [45] evaluates ChatGPT, a conversational language model, on bug fixing and code repairing in Python. The results show that ChatGPT is able to fix 19 out of 40 simple bugs, comparable to other state-of-the-art solutions. However, while the results look promising, the queries are simple code snippets that have a few lines. It remains unclear how well ChatGPT can parse complex code context in large programs. Its performance in identifying vulnerable code (beyond simple syntactic and logic bugs) and providing secure code suggestions by itself could form an interesting research topic. Other large language model-powered code completion tools, such as Copilot, have also been published in recent years. These models are trained with a huge amount of source code without any program analysis preprocessing. Due to limited resources, we are unable to train comparable models from scratch on program analysis processed data. We leave those comparisons as our further work.

We summarize our major findings from experiments.

- Program analysis preprocessing is very important even with advanced embedding options. With all the embedding options (sequence-level embedding, token-level embedding, or one-hot encoding), program analysis makes big improvements in API completion accuracy. Without program analysis, the best accuracy on bytecode with the most advanced *byteBERT* is only 57.59%. With the API dependence graph construction, *depBERT* on dependence paths achieves the highest accuracy of 93.52% on the basic dataset.
- Applying token-level embedding in API completion task training makes substantial improvement on program analysis process code corpora. On slices and dependence paths, the LSTM models trained with the token-level embedding *slice2vec* and *dep2vec* show significant accuracy improvements by 12% and 5%, respectively, compared with the one-hot vectors.
- The accuracy improvement of sequence-level embedding (0.55% on average) is not obvious under the basic setting. Hence, we do not recommend sequence-level embedding in that case. Meanwhile, we observe more significant improvements (5.10%) in sequence-level embedding under the cross-app scenario when the task-specific data size is small (App set 6). Thus, we recommend it for cross-app scenarios with small task-specific data.

4.5 Analogy Tests of Token-level Embedding

We perform the analogy tests to intuitively show the quality of token-level embeddings. Besides the impact on downstream tasks, good embedding vectors should also reflect the semantics of a token and its relationship with other tokens. In natural language processing, the quality of embedding is usually evaluated through analogous pairs (e.g., $men - women \approx king - queen$) [31–33]. Therefore, following the practice in the natural language field, we design a few analogy tests to help understand the quality of API embeddings based on different program analysis methods. In our work, we define analogous pairs as two pairs of APIs or constants, (a and a') with (b and b'), having a high degree of relational similarity (i.e., analogous) in terms of some programming property. For Java cryptographic code, we identify four categories of analogous pairs as follows. We show examples in Table 11.

Table 11. Four categories of analogous pairs we define among API methods and constants. We give a representative example for each category, where two pairs (a and a' vs. b and b') have a high degree of relational similarity (i.e., analogous) in terms of some programming property. For each category, the number of analogies used in our top k evaluation (Table 12) is also shown.

Category		Examples of Analogous Pairs	# of analogies
Direct Dependency	a_1	`javax.crypto.KeyGenerator: javax.crypto.KeyGenerator getInstance(java.lang.String)`	4
	a'_1	`javax.crypto.KeyGenerator: void <init>(int)`	
	b_1	`java.security.KeyStore: java.security.KeyStore getInstance(java.lang.String)`	
	b'_1	`java.security.KeyStore: void load(java.io.InputStream,char[])`	
Semantic Symmetry	a_2	`javax.crypto.KeyGenerator: javax.crypto.KeyGenerator getInstance(java.lang.String)`	4
	a'_2	`javax.crypto.KeyGenerator: javax.crypto.SecretKey generateKey()`	
	b_2	`java.security.KeyPairGenerator: java.security.KeyPairGenerator getInstance(java.lang.String)`	
	b'_2	`java.security.KeyPairGenerator: java.security.KeyPair generateKeyPair()`	
Argument Symmetry	a_3	`"AES"`	4
	a'_3	`javax.crypto.KeyGenerator: javax.crypto.KeyGenerator getInstance(java.lang.String)`	
	b_3	`"RSA"`	
	b'_3	`java.security.KeyPairGenerator: java.security.KeyPairGenerator getInstance(java.lang.String)`	
Syntactic Variants	a_4	`javax.crypto.Cipher: byte[] doFinal(byte[])`	2
	a'_4	`javax.crypto.Cipher: int doFinal(byte[],int)`	
	b_4	`javax.crypto.Mac: byte[] doFinal(byte[])`	
	b'_4	`javax.crypto.Mac: void doFinal(byte[],int)`	

Direct Dependency. For two APIs where one always accepts the other's output, they form a pair having a direct dependency. For example, after a `KeyGenerator` instance is created by `KeyGenerator.getInstance(.)`, it always needs to be initialized through `KeyGenerator.init(.)`. The analogous relation could also be found between `KeyStore.getInstance(.)` and `KeyStore.load(.)` where the latter loads the required information to the `KeyStore` instance created by the former. We view the two pairs as analogous pairs under this category.

Semantic Symmetry. For two classes `KeyGenerator` and `KeyPairGenerator`, the former generates secret keys for symmetric cryptography while the latter generates keys for asymmetric cryptography. There is a symmetry relationship between their APIs. For example, they both have the APIs `getInstance(String)` to create instances and APIs to generate the key.

Argument Symmetry. There is an analogous relation between API - constant pairs. For example, symmetric cipher `"AES"` can be passed to `javax.crypto.KeyGenerator: javax.crypto.KeyGenerator getInstance(java.lang.String)` as an argument. For asymmetric ciphers, `"RSA"` and API `java.security.KeyPairGenerator: java.security.KeyPairGenerator getInstance(java.lang.String)` have a similar relation.

Syntactic Variants. Some APIs share the same name but differ in their full signatures. These APIs are functionally equivalent but have different types of arguments or return values. We name them *syntactic variants*. For example, there are several APIs with the same name `doFinal(.)` of the Java class `Cipher` and Java class `MAC`.

Based on the analogous pairs, we define 14 tests. We calculate the vector of the embedded object b' based on the other three vectors of a, a', and b. If the actual embedding vector of b' appears in the top k nearest list of the calculated one (ideal value of b'), we say this analogy achieves rank k. Examples of how to calculate rank k are shown in Figure 5. The results of the 14 tests for *dep2vec*, *slice2vec*, and *byte2vec* are listed in Table 12.

In this small-scale analogous pairs evaluation, *dep2vec* performs the best. *dep2vec* achieves the best rank 12 times of the 14 test cases. *slice2vec* does well in some cases but performs poorly in the syntactic variants category. This is likely because the syntactic variant APIs usually appear in different contexts in slices, making *slice2vec* fail to recognize their similarity. For other more complicated relationships like semantic symmetry or

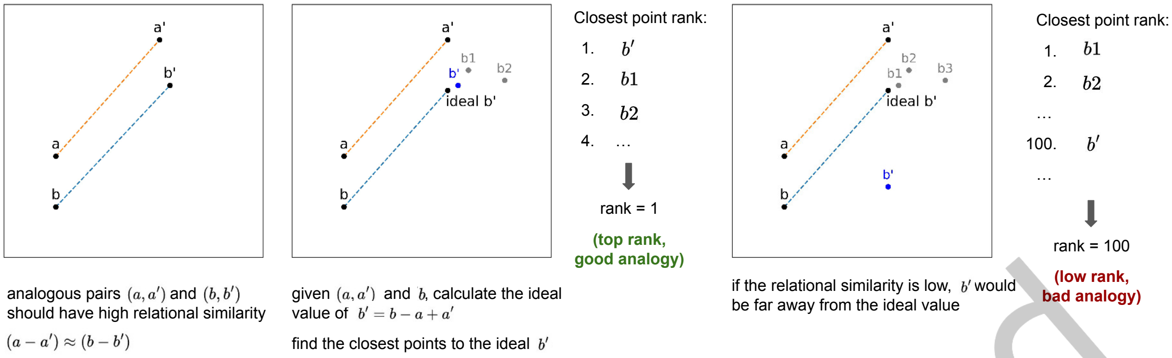

a'
b'
a
b
analogous pairs (a, a') and (b, b') should have high relational similarity
$(a - a') \approx (b - b')$
a'
b1
b2
b'
ideal b'
a
b
given (a, a') and b, calculate the ideal value of $b' = b - a + a'$
find the closest points to the ideal b'
Closest point rank:
1. b'
2. $b1$
3. $b2$
4. …
rank = 1
(top rank, good analogy)
a'
b2
b1
b3
ideal b'
a
b'
b
if the relational similarity is low, b' would be far away from the ideal value
Closest point rank:
1. $b1$
2. $b2$
…
100. b'
…
rank = 100
(low rank, bad analogy)

Fig. 5. Example of relational similarity of analogous pairs and how embedding rank is calculated. Generally, top rank implies the target embedding vector is of high quality, reflecting the semantics of the API well.

Table 12. The rank k of 14 analogous pairs in different embedding vectors. Smaller k suggests more accurate embedding vectors that better maintain analogous relationships. *dep2vec* outperforms others in most cases.

Category	**Rank** k (of correct vector)		
	dep2vec	slice2vec	byte2vec
Direct Dependency	**2**	**2**	50
	2	4	14
	3	13	41
	1	2	42
Semantic Symmetry	**2**	65	**2**
	20	**3**	8
	9	204	385
	4	239	355
Argument Symmetry	**1**	94	5
	1	49	16
Syntactic Variants	15	84	**2**
	9	95	249
	2	326	191
	9	278	419
Average	**5.7**	104	127

argument symmetry, the APIs and constants belonging to a pair often appear far away from each other in the code, increasing the difficulty of the test.

5 CASE STUDIES AND DISCUSSION

In this section, we provide a few case studies and discuss the practical design implications derived from our experiments.

5.1 Case Studies

To help interpret how program analysis and embedding vectors help API completion, we show several case studies.

Case Study 1. This case study is on the effectiveness of the API dependence graph construction. Figure 6(a) shows a slice-based test case that is mispredicted by both *slice2vec* and its one-hot baseline. For digest calculation, it is common for `MessageDigest.update(.)` to be followed by `MessageDigest.digest(.)`, appearing 6,697 times in

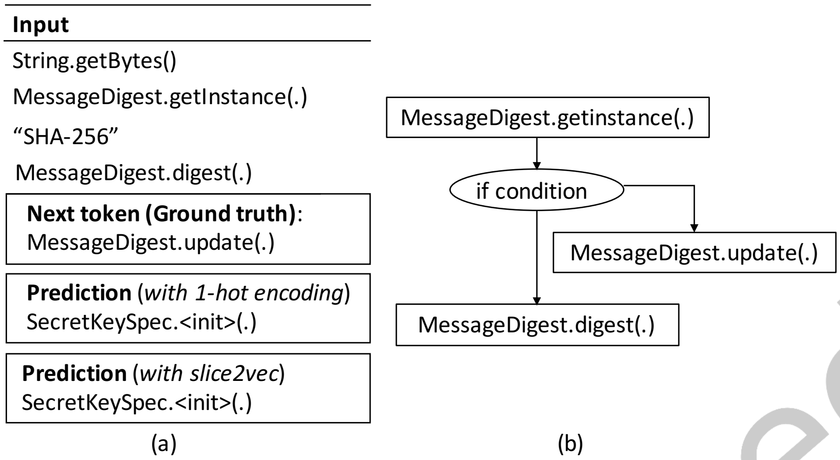


Fig. 6. Case Study 1. (a) A slice sequence that is predicted incorrectly. (b) The model trained on API dependence graphs makes the correct prediction.

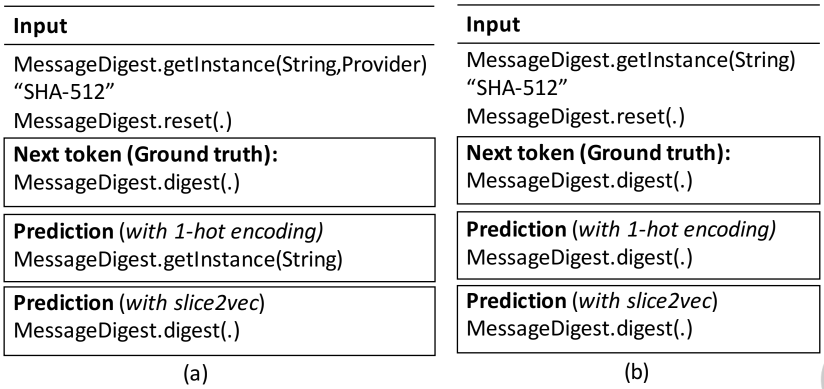
training. However, Figure 6(a) shows a reverse order, which is caused by the `if-else` branch shown in Figure 6(b). When `MessageDigest.update(.)` appears in an `if` branch, there is no guarantee which branch would appear first in slices. This reverse order is less frequent, appearing 1,720 times in training. Thanks to the API dependence graph construction, this confusion is eliminated, which predicts this case correctly.

Case Study 2. This case study is on the ability to recognize new previously unseen test cases. The slices in Figure 7(a) and Figure 7(b) slightly differ in the arguments of the first API. *slice2vec* makes the correct predictions in both cases, while its one-hot baseline fails in Figure 7(a). `MessageDigest.getInstance(String)` appears much more frequent than `MessageDigest.getInstance(String,Provider)` in our dataset. Specifically, the former API appears 207,321 times, out of which 61,047 times are followed by the expected next token `MessageDigest.digest(.)`. In contrast, the latter API – where one-hot fails – only appears 178 times, none of which is followed by `MessageDigest.digest()`. In *slice2vec*, the cosine similarity between `MessageDigest.getInstance(String,Provider)` and `MessageDigest.getInstance(String)` is 0.68. [7] This similarity, as the result of *slice2vec* embedding, substantially improves the model's ability to make inferences and recognize similar-yet-unseen cases.

5.2 Practical Design Implications

Our findings empirically demonstrate that, among various design choices, applying program analysis brings the most significant improvement in the cryptographic code completion task, up to 45.8%. While upgrading to larger models and increasing model sizes also boost the accuracy, the significance of improvements is not comparable to applying semantically meaningful preprocessing. We observe only a 2.45% improvement when upgrading the Transformer model size. While large language models substantially changed the natural language processing field, applying them to programming languages as is may not be ideal. As shown in section 4.2.2, pretraining provides only trivial enhancement to the model performance. This result suggests that instead of consuming high computational power for a slight boost, one should consider how to incorporate the most effective information for the prediction tasks.

[7] For one-hot vectors, this similarity is 0.



Input
MessageDigest.getInstance(String,Provider)
"SHA-512"
MessageDigest.reset(.)

Next token (Ground truth):
MessageDigest.digest(.)

Prediction (*with 1-hot encoding)*
MessageDigest.getInstance(String)

Prediction (*with slice2vec*)
MessageDigest.digest(.)

(a)

Input
MessageDigest.getInstance(String)
"SHA-512"
MessageDigest.reset(.)

Next token (Ground truth):
MessageDigest.digest(.)

Prediction (*with 1-hot encoding)*
MessageDigest.digest(.)

Prediction (*with slice2vec*)
MessageDigest.digest(.)

(b)

Fig. 7. Case Study 2. (a) A test case that is correctly predicted with *slice2vec*, but incorrectly with a one-hot vector. (b) A test case similar to (a), but both *slice2vec* and one-hot give the correct prediction.

Another key contribution of our paper is the comparison between different program analysis strategies. The evaluation reveals that the dependence path brings the best accuracy. A possible reason could be that the analysis goes beyond the method boundary and collects dependence paths across the entire program. In this way, all information related to the target API is preserved in the path and gets embedded into the input. Therefore, a key step in building a code completion model is to incorporate program analysis and the dependence path is a recommended method.

After deploying a code completion tool, a practical use scenario is to predict the next tokens on uncompleted programs. There exist more challenges when applying program analysis to code under development. Incorporating methods such as partial program analysis (PPA) [15, 16] is an important next step.

Soundness. Our conclusions are based on a rigorous approach with carefully controlled experiments. For the three dimensions, program analysis, token-level embedding, and sequence-level embedding, we measure the impact of a specific design by comparing the API completion trained with or without it.

Limitations. We briefly discuss our limitations and threats to validity. First, we perform security sanitization to filter insecure code in our dataset. However, security sanitization relies on a static analyzer that may not be perfect. Second, we apply static analysis to extract program slices and dependence paths. However, static analysis tends to overestimate execution paths. Thus, the slices and dependence paths used for learning might not necessarily occur. Third, we do not try other embedding techniques such as ELMo [40]. We met an incompatibility issue when adapting the published ELMo code for our API completion task. The published code requires outdated libraries such as TensorFlow v1.2, and CUDA 8 while our learning environment only supports CUDA 9 or later. We will consider adding more embedding models in the future. Lastly, our work focuses on Java cryptographic APIs. The generalizability to other languages, such as Python, is out of the scope of this work.

An internal threat to validity is that we use identical training hyperparameters for all the API completion experiments. When applying different program analysis and embedding techniques, we train neural networks with identical training hyperparameters. We did not tune hyperparameters to find the best practices for every case. An external threat to validity comes from the dataset we use in the measurement. We only perform API completion experiments with Java cryptographic API benchmark, although the embedding method is for general purposes. We choose Java cryptographic APIs because it is complicated and the code completion task is more challenging. Our future work will extend the benchmark with more diverse APIs to confirm our results.

6 RELATED WORK

There are two main branches of code embedding solutions.

Embedding without program analysis. First, a line of research develops pure data-driven solutions on general source code tokens without program analysis [3, 12, 14, 20, 26, 27, 46, 49]. They train neural network solutions to take as input programs that are treated as sequences of source code tokens. In [12], Buratti *et al.* claimed that the language model built on top of raw source code is able to discover abstract syntax tree (AST) features automatically.

Embedding with program analysis. Second, some studies (e.g., [4, 9, 23, 58]) leverage the program structural information through program analysis. For example, the authors of [58] learned code embedding after constructing the graph representations (e.g., control flow graphs, data flow graphs) of code. Hellendoor *et al.* [23] advocated a hybrid embedding method that considers both the graph structure and the raw sequences to overcome the size limit of graphs. To remove noises in code, Henkel *et al.* performed intra-procedural symbolic execution first and trained embedding vectors of symbolic abstractions from symbolic traces [24]. However, there have not been systematic studies on how various hybrid approaches compare with a pure data-driven approach or with each other, in terms of downstream task performance.

Since there are various program intermediate representations (IRs) under program analysis, the embedding objects also vary from approach to approach. For example, Henkel *et al.* obtained embeddings for self-defined symbolic abstractions. Ding *et al.* [18] obtained embedding vectors *asm2vec* for assembly code instructions. Ben-Nun *et al.* [7] embedded LLVM IR instructions of code. Although the idea of leveraging the program structural information in embeddings is identical, these embeddings for low-level instructions, or LLVM IRs cannot be directly compared with embeddings for API elements. Our *dep2vec* and *depBERT* can be viewed as graph-based embedding approaches applied to API elements.

A line of work focuses on API embeddings and related tasks [6, 11, 13, 19, 21, 35, 36, 56]. Our work also lies in this category. Nguyen *et al.* [35, 36] use API sequences in source code to produce embeddings for Java APIs and C# APIs. Using these vectors, they successfully mapped the semantic similar Java APIs with C# APIs. Our *byte2vec* can be viewed similarly to it as our API call sequences from bytecode are similar to their source code order. Chen *et al.* [13] trained the API embedding based on the API description (name and documents) and usage semantics. The obtained API embeddings are used to infer the likely analogical APIs between third party libraries. However, these solutions employ embeddings to help map analogical APIs, which is different from our task, API completion. In API completion work [34, 37, 38, 43, 47], there is either no discussion about the impacts derived from different embedding options.

7 CONCLUSION

Our measurement study, including the new benchmark, provides deep insights into the strengths and weaknesses of neural network techniques in the context of code completion. Our quantitative experimental results highlight the importance of program-specific analysis, which brings the most significant improvement for the code completion task, even with powerful data-driven deep learning approaches. The direct application of neural network approaches originally designed for natural languages may not give optimal accuracy, as programming languages have unique characteristics. Therefore, we emphasize the need for careful consideration and appropriate preprocessing when adapting natural language processing techniques for code-related tasks. Our ongoing and future work is on designing new neural architectures specific to software engineering tasks.

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation under Grant No. CNS-1929701 and the Virginia Commonwealth Cyber Initiative (CCI).

REFERENCES

[1] Sharmin Afrose, Ya Xiao, Sazzadur Rahaman, Barton P Miller, and Danfeng Yao. 2022. Evaluation of static vulnerability detection tools with Java cryptographic API benchmarks. *IEEE Transactions on Software Engineering* 49, 2 (2022), 485–497.

[2] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.

[3] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 38–49.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[6] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: a learnable representation of code semantics. In *Advances in Neural Information Processing Systems*. 3585–3597.

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. *arXiv preprint arXiv:1607.04606* (2016).

[9] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *International Conference on Learning Representations*.

[10] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[11] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: learning cross-language API mappings with little knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 796–806.

[12] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring Software Naturalness through Neural Language Models. *arXiv preprint arXiv:2006.12641* (2020).

[13] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. 2019. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering* (2019).

[14] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4818–4837.

[15] Barthélémy Dagenais and Laurie Hendren. 2008. Enabling Static Analysis for Partial Java Programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 313–328. https://doi.org/10.1145/1449764.1449790

[16] Barthélémy Dagenais and Laurie Hendren. 2008. Partial Program Analysis for Java. (2008). http://www.sable.mcgill.ca/ppa/

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[18] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489.

[19] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 745–759.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[21] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.

[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

[23] Vincent J Hellendoorn, C Sutton, Rishabh Singh, and P Maniatis. 2020. Global Relational Models of Source Code. In *International Conference on Learning Representations*.

[24] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 163–174.

[25] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of Tricks for Efficient Text Classification. *arXiv preprint arXiv:1607.01759* (2016).

[26] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *International Conference on Machine Learning*. PMLR, 5110–5121.

[27] Rafael-Michael Karampatsis and Charles Sutton. 2020. SCElmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214* (2020).

[28] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[29] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using Transfer Learning for Code-Related Tasks. *IEEE Transactions on Software Engineering* (2022).

[30] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango-Argoty. 2018. Secure coding practices in Java: Challenges and vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 372–383.

[31] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[32] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[33] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. 2013. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 746–751.

[34] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 511–522.

[35] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 756–758.

[36] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 438–449.

[37] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. 383–392.

[38] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2016. Learning API usages from bytecode: a statistical approach. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 416–427.

[39] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[40] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365* (2018).

[41] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[42] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2455–2472.

[43] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.

[44] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2020. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[45] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. https://doi.org/10.48550/ARXIV.2301.08653

[46] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[47] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2727–2735.

[48] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[49] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2291–2302.

https://doi.org/10.1145/3510003.3510621

[50] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[52] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2019. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations (ICLR)*.

[53] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 32 (mar 2022), 58 pages. https://doi.org/10.1145/3485275

[54] Ya Xiao, Wenjia Song, Jingyuan Qi, Bimal Viswanath, Patrick McDaniel, and Danfeng Yao. 2023. Specializing Neural Networks for Cryptographic Code Completion Applications. *IEEE Transactions on Software Engineering* (2023).

[55] Ya Xiao, Yang Zhao, Nicholas Allen, Nathan Keynes, Danfeng Yao, and Cristina Cifuentes. 2023. Industrial experience of finding cryptographic vulnerabilities in large-scale codebases. *Digital Threats: Research and Practice* 4, 1 (2023), 1–18.

[56] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API method recommendation via explicit matching of functionality verb phrases. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1015–1026.

[57] Danfeng Daphne Yao, Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Miles Frantz, Ke Tian, Na Meng, Cristina Cifuentes, Yang Zhao, Nicholas Allen, et al. 2022. Being the developers' friend: Our experience developing a high-precision tool for secure coding. *IEEE Security & Privacy* 20, 6 (2022), 43–52.

[58] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*. 10197–10207.

[59] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs.

A HYPERPARAMETER SELECTION IN OUR MEASUREMENT STUDY

There are many hyperparameters in our measurement study. Within each comparison group, we keep identical hyperparameters to guarantee a fair comparison.

A.1 Hyperparameters for training LSTM

With specific embeddings, we need to train a neural network model (e.g., LSTM or Transformer) to perform API completion. For those comparisons, we choose the number of epochs, learning rate, and batch size through some preliminary experiments. We train the LSTM model with slice2vec with different batch sizes, and learning rates, and check their accuracies within 20 epochs.

Table 13. The prediction accuracies obtained by LSTM with slice2vec with different hyperparameters.

Batch size	Learning rate	Epoch	Accuracy
1024	0.001	1	0.47
512	0.001	1	0.48
1024	0.01	1	0.48
1024	0.1	1	0.49
1024	0.001	10	0.83
1024	0.001	20	0.83

Based on our observation in Table 13, the accuracy differences between batch sizes 1024 and 512 are slight. We use batch size 1024 to reduce the training time. We compare the LSTM trained with learning rates of 0.001, 0.01, and 0.1. Their difference is also small. Therefore, we use batch size 1024 and a learning rate of 0.001 for all of the training tasks for API completion. For epoch, we found that the accuracy improvement after epoch 10 is negligible.

A.2 Hyperparameters for GraphCodeBert and CodeBert Training

We finetune the pretrained model GraphCodeBert and CodeBert on our dataset. To determine the batch size, we try different batch size options in finetuning GraphCodeBert on our dataset 6. The accuracy results after 10 epochs are shown in Table 14.

Table 14 suggests that a smaller batch size could result in higher accuracy. When we decrease the batch size from 512 to 16, the prediction accuracy keeps increasing. However, it also shows that batch size 16 is good enough as smaller batch size 8 results in no improvement. Therefore, we apply batch size 16 to all the comparative experiments on GraphCodeBert and CodeBert.

Table 14. The prediction accuracies obtained by GraphCodeBert finetuned with different batch size options.

Batch size	Epoch	Accuracy
512	10	0.38
256	10	0.48
128	10	0.51
64	10	0.53
32	10	0.54
16	10	0.55
8	10	0.55

A.3 Impact of different design choices on prediction accuracy

We report the improvement brought by different design choices, including embedding strategies, program analysis preprocessing, and model size, on the API completion task prediction accuracy (Figures 8, 9, and 10).

A.4 Applying Deep Learning to Software Engineering Checklist

In this section, we provide details of our design choices for deep learning models, following the DL4SE guidelines in [53].

Step 1: Preprocessing and Exploring Software Data. We extract API sequences from 16,048 Android apps for our experiments. Our program analysis preprocessing approaches (i.e., bytecode, program slices, and dependence graph) yield 708k, 927k, and 566k sequences, respectively (section 3.4). The data size is sufficient for large deep learning models to learn from. We use 80% of the data for training and 20% for testing.

Step 2: Perform Feature Engineering. We use three different strategies to extract API sequences from Android apps and train embedding vectors representing the API. The corresponding API embedding sequences are used as input to the model. The dataset is labeled as we have the ground truth for our API prediction tasks. For the token prediction task, the ground truth is the last API token in the sequence. For the sequence prediction task, the ground truth is the API sequence following the input sequence. Examples of API sequences generated from the three preprocessing strategies are shown in Figure 1.

Step 3: Select a Deep Learning Architecture. Because of the sequential nature of the data, we use deep learning models LSTM and Transformer in our experiment. Both models have been used for code completion tasks in previous works. We train the models for 10 epochs because, after 10 epochs, the accuracy improvement from each additional epoch is less than 0.001. We provide details about the hyperparameters used for each experiment in their corresponding section (i.e., section 4.1 for RQ1, 4.2 for RQ2, 4.3 for RQ3, and 4.4 for RQ4).

Step 4: Check for Learning Principles. Our data are composed of Java cryptographic APIs extracted from Android apps, covering 3,739 unique APIs from various libraries. This variety provides enough representation of

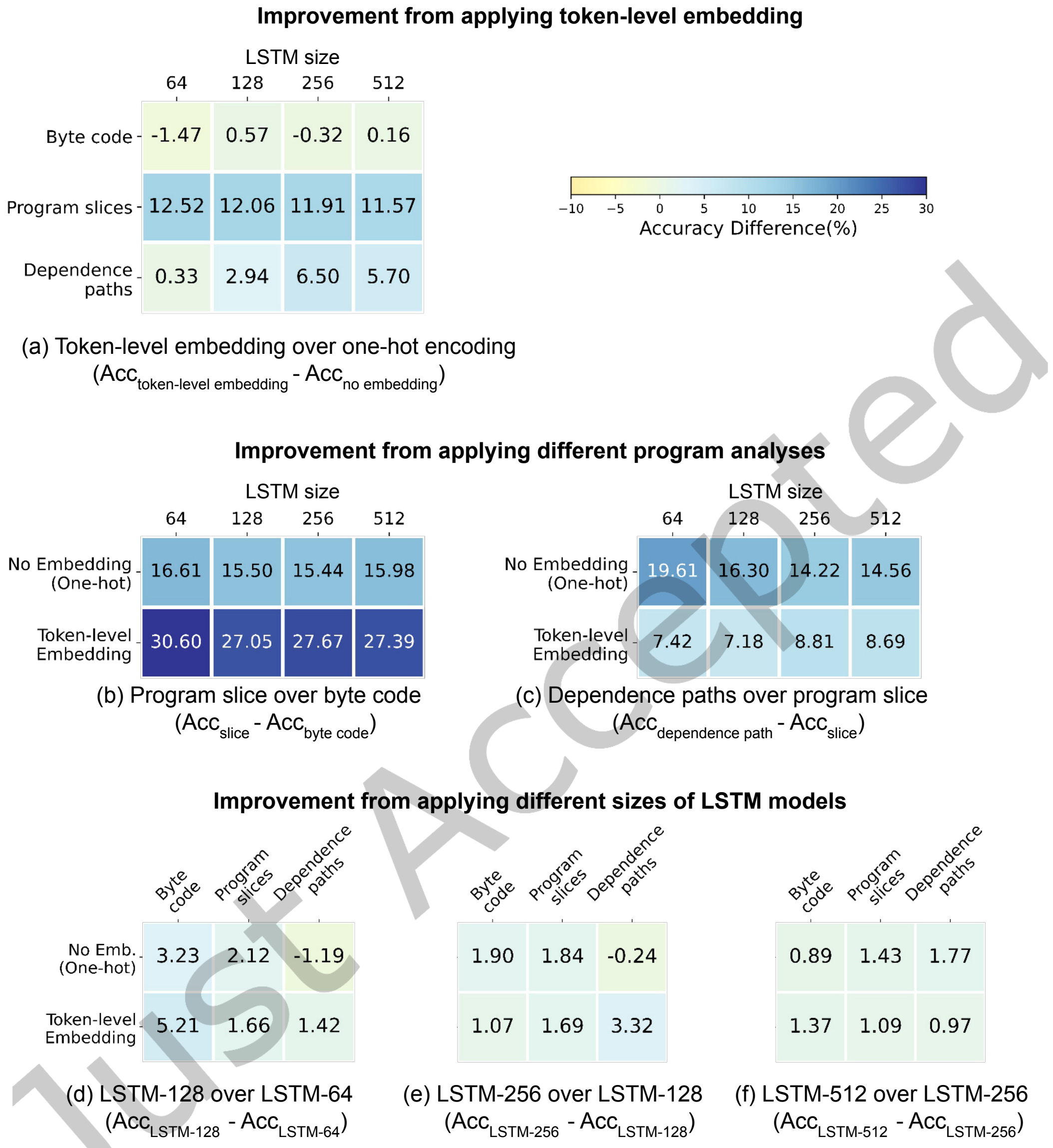
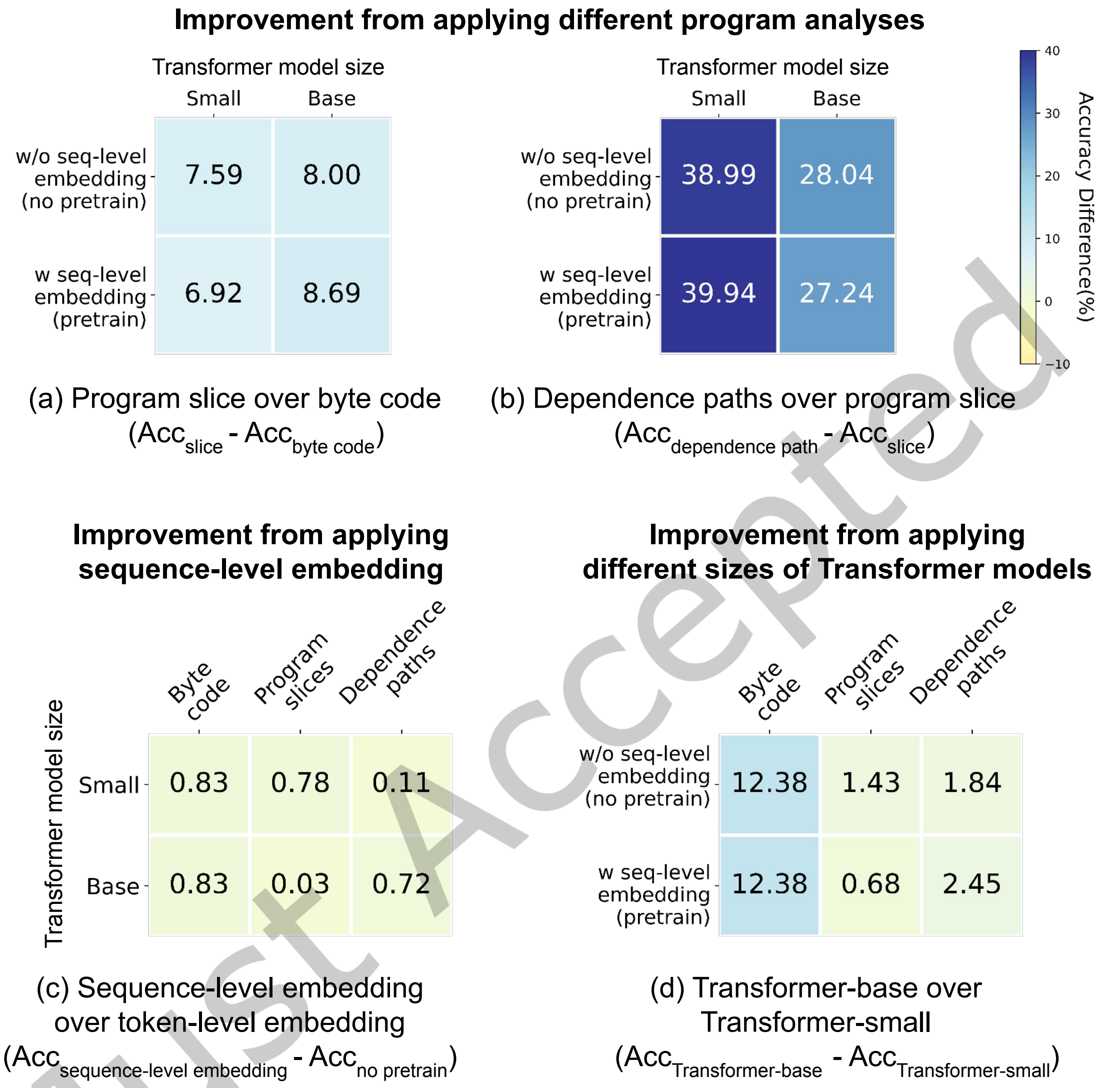


Fig. 8. The accuracy differences of token-level embedding, program analysis preprocessing, and increasing model sizes. (a) shows the accuracy difference of token-level embedding, which are the accuracies achieved with token-level embedding minus the accuracies achieved without token-level embedding. (b) shows the accuracy difference of program slicing, which is the accuracies achieved on program slices minus the accuracies achieved on bytecode sequences. (c) is the accuracy difference of API dependence graph construction, which is the accuracies on API dependence paths minus those on program slices. (d) (e) (f) are accuracy differences of increasing the LSTM hidden vector size from 64 to 128, from 128 to 256, and from 256 to 512, respectively.



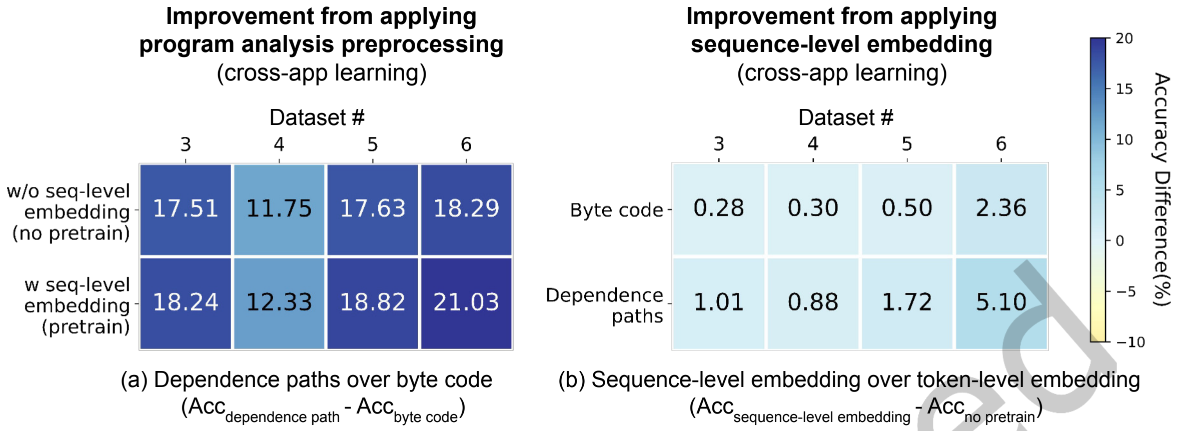
(a) Program slice over byte code (Acc_{slice} - $Acc_{byte\ code}$)

(b) Dependence paths over program slice ($Acc_{dependence\ path}$ - Acc_{slice})

(c) Sequence-level embedding over token-level embedding ($Acc_{sequence\text{-}level\ embedding}$ - $Acc_{no\ pretrain}$)

(d) Transformer-base over Transformer-small ($Acc_{Transformer\text{-}base}$ - $Acc_{Transformer\text{-}small}$)

Fig. 9. The accuracy differences of sequence-level embedding, program analysis preprocessing, and increasing model sizes. (a) shows the accuracy difference of slicing, which are the accuracies achieved on program slices minus the accuracies achieved on byte code sequences. (b) is the accuracy difference of API dependence graph construction, which is the accuracies on API dependence paths minus those on program slices. (c) shows the accuracy difference of applying sequence-level embedding, and (d) is the accuracy differences between using Transformer-small and Transformer-base neural networks.



(a) Dependence paths over byte code ($Acc_{dependence\ path}$ - $Acc_{byte\ code}$)

(b) Sequence-level embedding over token-level embedding ($Acc_{sequence\text{-}level\ embedding}$ - $Acc_{no\ pretrain}$)

Fig. 10. The accuracy differences of sequence-level embedding, and program analysis preprocessing one new App sets 3, 4, 5, 6. (a) shows the accuracy difference of program analysis preprocessing (program slicing + API dependence graph construction), which are the accuracies achieved on API dependence paths minus the accuracies achieved on bytecode sequences. (b) is the accuracy difference of sequence-level embedding, which is the accuracies achieved with sequence-level embedding minus the accuracies achieved without sequence-level embedding.

cryptographic API usage. We report the efficiency of program analysis-aided embedding through comparison with the naive approach (i.e., bytecode).

Step 5: Check for Generalizability. Our experiments are conducted on API sequences extracted from 12 different categories, covering a wide range of Android apps in practice. This proves that our results are generalizable to apps for diverse purposes. To confirm our models are not overfitted to the apps used in training, we further conduct a cross-app evaluation (i.e., 20% of apps are for testing only). The model accuracies in the cross-app setting are comparable with our basic setting, verifying there is no overfitting. Lastly, we compare our embedding approaches with the state-of-the-art models, namely CodeBERT and GraphCodeBERT, on the same dataset using the same metric to support our results.